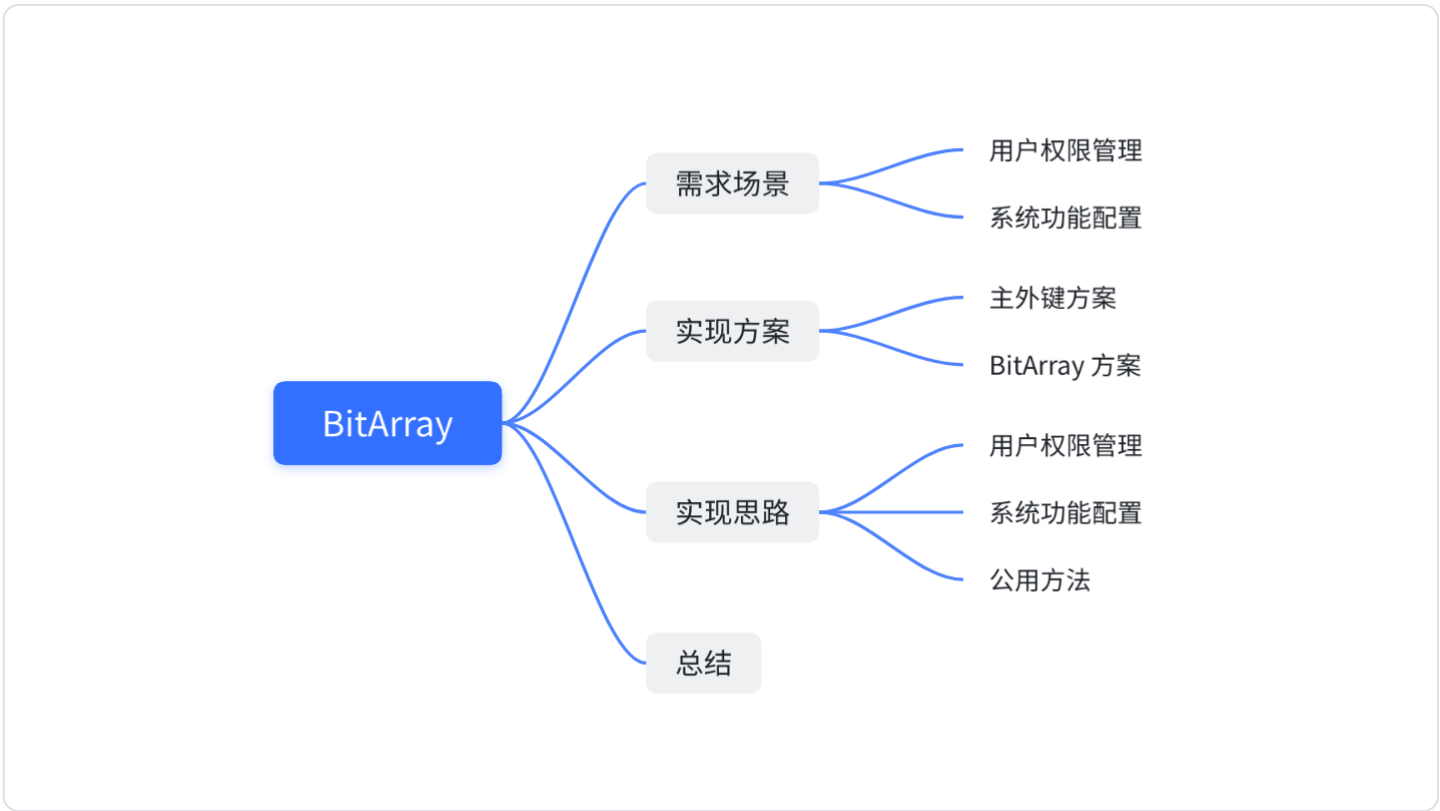
基于位运算的权限控制

本文从一个具体需求场景入手，对比两种权限控制系统的实现方案，介绍基于位运算的权限控制。

文章结构如下：



一、需求场景

系统目前需要以 RBAC (Role Based Access Control) 方式控制不同用户的权限，同时还需要功能配置系统满足不同团队定制化需求。

上述需求共分为**两**个目标：

1. **用户权限管理**

同一团队内不同用户能够使用的功能集合，同样需支持团队内自行配置，仅当该用户在此团队时生效，对该团队其他用户、该用户其他团队不会有影响。依次可以进行团队内个人维度的功能管控。

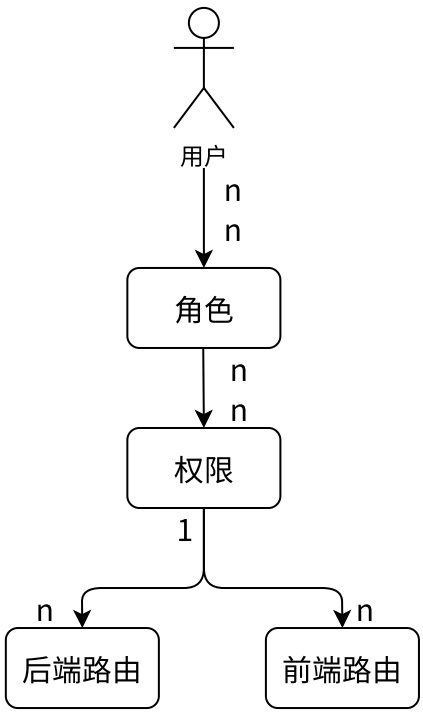
2. **团队功能配置**

不同团队的所有成员能够使用的功能集合，需要支持动态配置。此配置仅对该团队生效，且对该团队所有成员均生效。以此为基础可以为不同团队提供不同的系统功能购买方案。

> 以上两目标逻辑上独立，实现方式存在共通之处。

二、实现方案

此部分分别介绍传统主外键方案与 BitArray 方案，讲解两种实现的思路，从存储、运算等角度进行比对。两种方案都以单表的形式存储权限的基础信息，主要区别在于**角色和权限的关联方式**，以及角色表结构的设计。

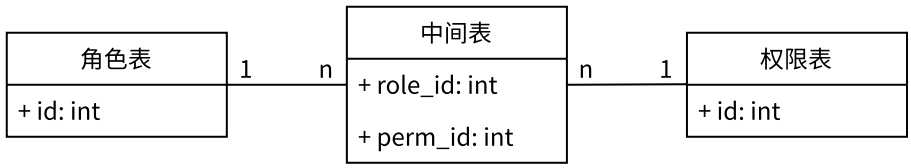


> 角色与权限通常是多对多关系，两种方案主要区别在于这一步的映射在数据表层面上如何完成。
>
> 权限与路由的对应关系既可以使用一对多：一个权限控制多个路由、多个权限控制一个路由等，也可以使用一对一关系。这部分不是本文主要讨论内容，因此这里默认单个权限对应多个后端路由或

> 前端路由。

中间表方案

使用一张角色&权限中间表控制二者关系。这里举出一个较为简单的设计方式。



这种设计下，角色到权限的检索需要查询两次数据库：查询中间表获取该角色拥有的所有权限 id 列表；根据权限 id 列表查询所有权限。

在用户量不大的场景中（角色表与权限表均为百量级），这种方案较为简单，效率可以接受。不过即便角色与权限均是百量级，中间表的数据量也是万量级了。这种 m*n 的表数量关系（角色表 m，权限表 n），对于权限系统这样一个如空气般无处不在的基础设施类系统而言，效率上还是有些捉襟见肘。

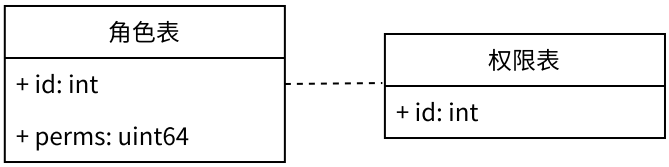
这种方案也有一个显著的优点，就是扩展性不错。若系统对性能要求不是很苛刻，或能解决读取速度上的问题，那不论是十种权限 / 角色还是上万个角色 / 权限，代码实现上都不需要进行修改。

📌 **优点**：简单；扩展性好；数量级不大时效率可以接受。

缺点：中间表随角色、权限表数据增多急剧膨胀。

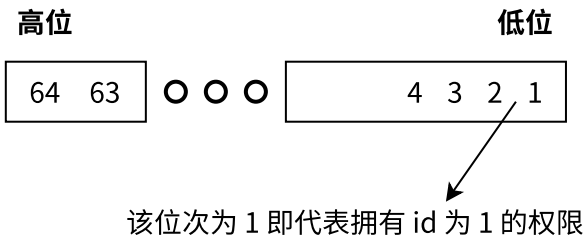
BitArray 方案

不使用中间表，直接在角色表中记录二者的关系。



角色表中 perms 字段类型为 uint64，共 64 个比特位，每一个二进制位代表一个独立权限（也可以使用字符串，公用方法实现上有细微差异）。这种约束性的映射关系依照由低到高或由高到低的次序人为制定。下图以低位优先为例。

> 这里的低位指数字层面上的低位（左高右低），而非内存层面上的高位低位。



这种设计用人为设定的映射关系来发挥中间表的作用，很大程度上优化了存储效率。角色到权限表的过程，只需要计算角色表中 perms 字段代表的权限 id 列表，然后再去查询即可，整个过程只需要一次查询。在存储空间上，也节省了 m*n 的存储量，只需要存储 m + n 条记录（前一种方案为 m*n + n + m）。角色表中只需要一个字段，即可表示 64 种甚至更多权限的持有状态。

但这样也会带来一些问题，主要有以下三方面：

1. **代码实现较为复杂**

   代码逻辑上需要实现由 uint64 向 id 列表的映射，以及反向转换。

2. **可扩展性较差**

   随着权限个数的增加，角色表需要不断新增 perms2、perms3 字段来满足更多需求，同时相关的业务代码也需要进行修改，兼容多个权限字段的转换逻辑。

3. **可能会出现数据不一致的问题**

   角色表与权限间不存在数据层面的强关联，因此权限 id 相关的变更可能没办法同步至角色，会导致角色的权限信息失真出现偏差，需要在代码逻辑上对此进行规避。

> 考虑到运算、存储效率上的提升，在一些场景下这种方案还是有着较大优越性的。

✏️ **优点**：运算、存储效率高；有设计感。

缺点：实现复杂；扩展性差；可能出现数据不一致。

三、实现思路

这部分以操作步骤的形式描述实现思路，分别介绍用户权限管理与团队功能配置，最后列出几个以 Go 语言实现的 BitArray 公用处理方法。

用户权限管理

用户维度，判断用户的前、后端路由信息。

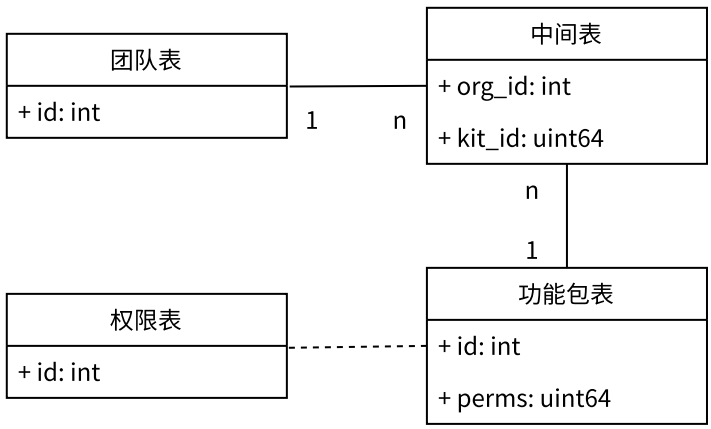
1. 根据用户 id 查询该用户的所有角色中间表；
2. 取出该用户的所有角色 id 列表；

3. 根据角色 id 列表获取角色信息；
4. 遍历角色，取出 perms 字段进行并集运算，获得最终 BitArray，转化为权限 id 列表；
5. 根据权限 id 列表获取所有权限信息（依此进行前后端路由的鉴权）。

团队功能配置

不同团队所能使用系统功能的范围不同，重点在于功能过滤，主要场景是在用户的权限控制之前（或之后）使用该团队功能全集进行一次过滤。

此处新增一个实体：功能包。结构与角色表类似，表示原子权限的集合，与权限表的关联也通过 BitArray 实现。团队与功能包的关联通过中间表实现，一个团队功能的全集是其拥有的所有功能包的总和。



1. 根据团队 id 查询该团队的所有功能包中间表；
2. 取出该团队的所有功能包 id 列表；
3. 根据功能包 id 列表获取功能包信息；
4. 遍历功能包，取出 perms 进行并集运算，获得最终 OrgBitArray；
5. 由用户权限管理的流程获取到用户的 UserBitArray；
6. 将 UserBitArray 同 OrgBitArray 作与运算，将结果转化为权限 id 列表，即为过滤后的用户最终权限。

公用方法

BitArray 常用操作较为固定，此处给出 6 个通用实现，以 Go 语言为例：

1. Id 数组转化为 BitArray

```go
// ToInt 接收原子权限 Seq 的切片，根据位次转化为 BitArray.
// 最终计算 BitArray 对应的 int 列表并返回.
func ToInt(permSeqs []int) []uint64 {
    var bitArray1 uint64
```

```
    var bitArray2 uint64
    for _, seq := range permSeqs {
        if seq != 0 {
            if seq <= 64 {
                bitArray1 |= 1 << (seq - 1)
            } else {
                bitArray2 |= 1 << (seq - 64 - 1)
            }
            // TODO 第三次扩展.
        }
    }
    return []uint64{bitArray1, bitArray2}
}
```

2. BitArray 转化为 id 数组

```
// ToSlice 接收 BitArray 对应的 int 值，根据位次解析原子权限 Seq.
// 最终以原子权限 Seq 列表的格式返回.
// 数组中 bitArray 权重依 index 升高，0 位首个 bitArray 代表 1~64 原子权限，1 位代表
// 最终返回结果会去除重复 seq，并且会升序排列.
func ToSlice(bitArrays []uint64) []int {
    var permSeqs []int
    for index, bitArray := range bitArrays {
        var flag uint64 = 1
        for i := 1; i <= 64; i++ {
            if bitArray&flag != 0 {
                permSeqs = append(permSeqs, i+index*64)
            }
            flag = flag << 1
        }
    }
    return permSeqs
}

```

3. BitArray 取并集

```
// Union 接收多个 BitArray 组成的切片,将其去重取并集后返回一个 BitArray.
func (bitArrayManager) Union(bitArrays []uint64) uint64 {
    if len(bitArrays) == 0 {
        return 0
    }
    var permBitArray uint64
```

```
    for _, bitArray := range bitArrays {
        permBitArray |= bitArray
    }

    return permBitArray
}

```

4. BitArray 取交集

```
// Intersect 接收多个 BitArray 组成的切片，将其去重取交集后返回一个 BitArray.
func (bitArrayManager) Intersect(bitArrays []uint64) uint64 {
    if len(bitArrays) == 0 {
        return 0
    }
    var permBitArray uint64 = 0xffffffffffffffff
    for _, bitArray := range bitArrays {
        permBitArray &= bitArray
    }

    return permBitArray
}
```

5. 判断两个 BitArray 是否存在包含关系

```
// Include 接受两个 BitArray，判断 a 是否包含 b 的所有原子权限.
// 若全部包含则返回 true，否则返回 false.
func (bitArrayManager) Include(a uint64, b uint64) bool {
    return b == a&b
}

```

6. 从 BitArray 中删除某个 id 的权限

```
// Delete 将一个原子权限 BitArray 取出部分权限.
// 接收 BitArray 列表和一个原子权限 Seq 列表.
// 从前者中取出后者.
func (bitArrayManager) Delete(bitArrays []uint64, seqs []int) []uint64 {
    if len(bitArrays) == 0 {
        return bitArrays
    }
```

```
    seqBitArrays := BitArray.ToInt(seqs)
    for index, bitArray := range bitArrays {
        bitArrays[index] = bitArray & ^seqBitArrays[index]
    }
    return bitArrays
}

```

四、总结

上述所讲的例子都是最基本的脚手架，只是抛砖引玉，简单介绍一个基于位运算的方案设计。这个设计在应用于实际业务开发时还需考虑很多数据、代码层面上的细节问题，本文不再作过多赘述。

位运算运算效率较高，如果能找到合适的业务场景进行应用，可以很大程度上提升程序的整体运行效率。对于权限、功能管控这类的基础设施类服务，越少的存储空间、越快的运行速度就意味着更低的存在感，减少用户的感知度，提升体验。

位运算在业务中的应用场景肯定不止于此，如 ai_process 等，更多优秀设计还有待不断发掘。